AFRL-IF-RS-TR-1999-6
Final Technical Report
January 1999

DISTRIBUTED EVENTS IN SENTINEL: DESIGN AND IMPLEMENTATION OF A GLOBAL EVENT DETECTOR

University of Florida

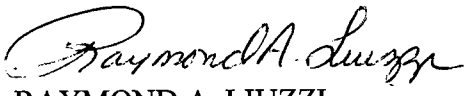
S. Chakravarthy, H. Liao, and H. Kim

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1999-6 has been reviewed and is approved for publication.

APPROVED: 

RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: 

NORTHRUP FOWLER III, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1999	Final Aug 96 - Oct 98

4. TITLE AND SUBTITLE

DISTRIBUTED EVENTS IN SENTINEL: DESIGN AND IMPLEMENTATION OF A GLOBAL EVENT DETECTOR

5. FUNDING NUMBERS

C - F30602-96-1-0275
PE - 62232N & 62702F
PR - R427
TA - 00
WU - P9

6. AUTHOR(S)

S. Chakravarthy, H. Liao, and H. Kim

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Florida
Department of Computer and Information Science
P.O. Box 116120
Gainesville FL 32611-6120

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/IFTB
525 Brooks Road
Rome NY 13441-4505

NAVY/NCCOSC RDTED44208
53245 Patterson Road
San Diego CA 92152-7151

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1999-6

11. SUPPLEMENTARY NOTES

This effort was jointly funded by AFRL/IFTB (formerly Rome Laboratory) and NAVY/NCCOSC (Ms. Leah Wong).
Air Force Research Laboratory Project Engineer: Raymond A. Liuzzi/IFTB/(315) 330-3577

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In conventional database management systems, data are created, updated, retrieved and deleted in a passive way; that is, only in response to operations performed by users or application programs. Active database management systems (ADBMSs) enhance the functionality of conventional database systems by issuing operations on their own in response to event occurrences or satisfied conditions. ECA (event-condition-action) rules are used to capture this activity capability: when an event is detected, a condition is checked and an action is executed if the condition is satisfied. Many computing applications are distributed in nature and hence require support for distributed computing. Many of the active OODBMS developed recently do not address event specification outside of their address space. Rules cannot be specified on events that occur in one or more applications. That is, none of them addresses processing ECA rules in a distributed environment. This report extends the earlier work on Sentinel and APBMS, by supporting global event specification and detection. Global events definitions are extended to Snoop, and a global event detector (GED) is implemented to detect events that span multiple applications.

14. SUBJECT TERMS

Database, Knowledge Base, Artificial Intelligence, Software, Computers

15. NUMBER OF PAGES

56

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Figures

Abstract

In conventional database management systems, data are created, updated, retrieved and deleted in a passive way; that is, only in response to operations performed by users or application programs. This *demand-based* mechanism cannot meet the requirement of many nontraditional applications that need either to monitor changes to the database state or recognize certain happenings or events of interest (some may occur outside the purview of the DBMSs), and initiate appropriate actions without user or application intervention. Active database management systems (ADBMSs) enhance the functionality of conventional database systems by issuing operations on their own in response to event occurrences or satisfied conditions. ECA(*event-condition-action*) rules are used to capture this active capability: when an event is detected, a condition is checked and an action is executed if the condition is satisfied. Rule definition, event detection and action execution are some of the new features provided by an ADBMS.

Many computing applications are distributed in nature and hence require support for distributed computing. Many of the active OODBMS developed recently do not address event specification outside of their address space. Rules cannot be specified on events that occur in one or more applications. That is, none of them addresses processing ECA rules in a distributed environment.

This report extends the earlier work on Sentinel by supporting global event specification and detection in a distributed database system. Global events definitions are extended to Snoop, and a global event detector (GED) is implemented to detect events that span multiple applications.

*This work was supported by the Office of Naval Research and the SPAWAR System Center – San Diego and by the Rome Laboratories under contract No. F30602-96-1-0275

1 Introduction

In conventional database management system, data are created, updated, retrieved and deleted in a passive way; that is, only in response to operations performed by users or application programs. This *demand-based* mechanism cannot meet the requirements of many nontraditional applications that need either to monitor changes to the database state or recognize certain happenings or events of interest that occur outside the purview of the DBMSs, and initiate appropriate actions without user or application intervention. Active database management systems (ADBMSs) enhance the functionality of conventional database systems by issuing operations (defined by the application designers) on its own in response to event occurrences or satisfied conditions. ECA (*event-condition-action*) rules are used to capture this active capability: when an event is detected, a condition is checked and an action is executed if the condition is satisfied. Rule definition, event detection and action execution are some of the new features provided by an ADBMS.

Much of the earlier research on active databases focus on the support of active capability in the context of centralized DBMS: both relational and object-oriente. In the following section, several active OODBMS implementations are introduced and compared.

1.1 Previous Research On Active OODBMS

1.1.1 HiPAC

HiPAC [1] is an object-oriented active database system which provides active database capabilities by means of ECA rules. HiPAC proposed three types of *primitive* events:

1. Data manipulation events, which are related to a method execution on objects. Two event modifiers *begin of* and *end-of* are used to transform an arbitrary interval into two logical events.

2. Clock events, which can be absolute, *relative* or *periodic*. An absolute event refers to a specific time point (e.g., 2:30:30.00,Nov 24 1996). A *relative* event specifies a temporal offset to some reference events (e.g., 45 seconds after event E1 occurs). A *periodic* event is specified by a reference event and a time period which is to be signaled periodically.

3. External events, which are specified and raised explicitly by users or application programs.

Composite events supported by HiPAC are composed using three operators: *disjunction*, *sequence* and *closure*.

1. A *disjunction* of two events E1 and E2 is raised when either E1 or E2 is signaled.

2. A *sequence* of two events E1 and E2 is raised if event E1 happened before event E2 is signaled.

3. A *closure* of an event E1, which is denoted as [E1*; E2] is raised when E1 has been signaled an arbitrary number of times before event E2 is signaled.

1.1.2 Ode

Ode [2] is an OODBMS which incorporates rules in the form of *constraints* and *triggers* to support active capabilities. *Constraints* are used to maintain database integrity and *triggers* are used for monitoring database conditions which are applicable only to the instances specified by the user at run time. Instead of ECA rules, Ode supports EA (event and action) rules which incorporate condition with the event part.

Four kinds of basic events are supported by Ode: 1. object state events, which are raised after the state of an object is changed through a public member function. e.g., after an object is created, before or after it is updated; 2. method execution events, which are signaled before or after a method is executed on an object; 3. time events, which are similar to clock events proposed in HiPAC; 4. transaction events, that are signaled at the beginning or at the end of a transaction, begin or end of a commit, or begin or end of an abort.

Since Ode merges condition with event part, these basic events become *primitive* events only when they meet a condition requirement, that is, they are qualified with a mask.

An *event history* is used to represent the semantics of *composite* events. Four event operators are used to support *composite* event definition:

1. A conjunction (^) of two events E1 and e2 is signaled when both events happen at the same time in the *event history.*

2. A not (!) operator of an event E1 (!E1) is used to denote event occurrences in the *event history* at which E1 is not signaled.

3. A *relative* of two events E1 and E2 is raised when event E1 happens before event E2 occurs.

4. A *relative+* operator of event E is used to present the *closure* of *relative* (E,E), which is signaled whenever E or an arbitrary number of successive E occurs.

The event detection in Ode is implemented using a finite state automata. Each event expression maps an *event history* to another event history that contains only those events at which the event expression is satisfied when the event is triggered.

1.1.3 ADAM

ADAM [3] is an active OODBMS implemented in PROLOG. It supports ECA rules in which both events and rules are treated as first class objects.

Events are defined with an event class hierarchy. Three subclasses are derived from a base event-class: *viz, db-event, clock-event* and *application-event.* An event is presented as an instance of one of these subclasses. An event is generated before or after a method is executed on an object.

In each class definition, the *class-rules* attribute is extended to indicate which rules to check when the object raises an event. Each rule is an instance of a *Rule* class and rule operations are

implemented as class methods. The *Rule* class consists of six attributes: *event, active-class, is-it-enabled, disabled-for, condition* and *action. Event* attribute indicate the event from which this rule is triggered. *Active-class* is the name of the class on which the rule is applicable. Whether the rule is triggered or not is specified by *is-it-enabled* attribute. *Disable-for* indicate the set of instances for which the rule is disabled. *Condition* and *action* present the rule's condition and action part.

When an event is raised, all the method's arguments are passed by the system to the condition and action part of the rule.

1.1.4 SAMOS

SAMOS [4] is an active OODBMS. It support five kinds of *primitive* events which are *method events, value events, transaction events, time events* and *abstract events.* Six operators are provided to compose *composite* event specifications, which are, *conjunction, sequence, disjunction, last, TIMES,* and *NOT.* A Colored Petri Nets which is called SAMOS Petri Nets is used to detect *composite* events. A Petri Net consists of *places, transitions, arcs. Arcs* connect *places* with *transitions* and *transitions* with *places.* The *places* correspond to the states of the Net, and such states may be changed by the *transitions* that correspond to the events which may occur. In Petri Nets, *tokens* represent event occurrences and capture the event type and the event parameters. A *place* in a Petri Net contains tokens of one specific token type. When an event occurs, a corresponding token is inserted into all *places* representing its event type. A *transition* can fire if all its input *places* contain at least one token. Then one token will be removed from each input *place* and inserted into each output *place.* Inserting a token into an end *place* corresponds to the detection of a composite event. The event parameters are part of the token.

1.1.5 Sentinel

Sentinel is an active object-oriented database management system which support ECA rules in both centralized and now in a distributed environment. An event specification language Snoop has been developed to specify events which include local events (local *primitive* events and local *composite* events) as well as global events (global *primitive* events and global *composite* events). Two event detection mechanisms, namely, a *local event detector* and a *global event detector*, are implemented to monitor the behavior of local events as well as global events across applications.

Three types of *primitive* events are supported by Sentinel:

1. Database events, which correspond to database operations used to manipulate data. Every method of an object is a potential *primitive* event, and they are transformed into events using two event modifiers: *begin-of* and *end-of.*

2. Temporal events, which include *absolute* and *relative* temporal events. An *absolute* temporal event is specified as an absolute value of time. A *relative* temporal event is specified by a reference event and a time offset.

3. External events, which denote events defined by users or application programs and are registered with the system. They are also called global events which support ECA rules processing in a distributed system. External events are assumed to be detected outside the system but are signaled to the system along with their parameters.

Composite events are composed by applying a set of operators to *primitive* events and previously defined *composite* events. In order to support global event detection, a *composite* event definition allows combination of events from different applications.

Events and rules can be defined at the class level (inside a class definition) as well as at the instance level (outside of a class definition). A class level event and rule is applicable to every object of this class, whereas an instance level event and rule is applicable only to the specific object instance. Four parameter contexts, Recent, Chronicle, Continuous, and Cumulative are supported in Sentinel. The parameters of a *primitive* event are the parameters of the method that this *primitive* is declared on and the time of the event occurrence. The parameter of a *composite* event is the combination of parameters of its constituent events.

In Sentinel, multiple rule executions, nested rule executions and prioritized rule executions are supported. Two coupling modes, immediate and deferred are implemented.

1.2 Motivation

Four research prototypes discussed earlier (HiPAC, Ode, ADAM, SAMOS) support active database functionality in a centralized database system. Many of the active OODBMS developed recently do not address event specification outside of their address space. Rule cannot be specified on events that involve multiple applications. That is, none of them address processing ECA rules in a distributed environment.

Many computing applications are distributed in nature and hence require support for distributed computing. For example, Telecommunication applications are inherently distributed. A number of telephone companies are connected to the telephone network. Telephone subscribers rent their telephones from one specific company, but may have accounts with other companies as well. Three major telephone companies, AT&T, GTE, and BELL, offer a special service to customers having an account with each of them and paying their bills *promptly*. This *promptly* action needs to monitor events from these three distributed companies. Active OODBMS in a centralized system cannot meet this requirement. A mechanism is needed to support processing ECA rules in a distributed environment.

This report extends the earlier work on Sentinel by supporting global events specification and detection in a distributed environment. Global event definitions are added to SNOOP (an event specification language of Sentinel), and a global event detector (GED) is implemented to detect events that span multiple applications.

2 Related Work

In addition to Sentinel, several research efforts attempt to monitor the behavior of distributed systems.

- Microsoft's COM (component Object Model) [5] provides a basic event service that support only primitive event detection. Any action that changes control, e.g., Changes to data, changes to the views on data, renaming of objects, clicking the mouse can be treated as an event. There is no notion of composite event.

- CORBA [6] supports an event service called suppliers to notify event occurrences to consumer via an event channel. Suppliers and consumers communicate with each other through a single well-known event channel object. The push model and pull model are supported as event modification models. In the push model, the supplier initiates the transfer of event data to consumers. In the pull model, the consumer requests event data from suppliers. As in COM, no services are supported for composite events.

- In Schwiderski's thesis [7] , she presents a general solution to monitoring the behavior of distributed systems and proposes an approach to event-driven monitoring of distributed systems which provides the full functionality of event specification, event semantics, and event detection. The work is concerned with the syntax, the semantics, the detection, and the implementation of events in terms of physical time.

The syntax of primitive and composite events is derived from the work of both active database systems and distributed debugging systems. Primitive events can relate to physical time, to occurrences inside database systems or application programs, and to arbitrary external signals. In a distributed system, *primitive event types* are *site related.* A *primitive event expression* is a character string denoting either a time event, a data manipulation event, a transaction event, or an abstract event. A specified primitive event expression determines a *primitive event type.* A time event denotes the readings of a particular local clock and can be *absolute, relative, or periodic.* A data manipulation event relates to the *insertion, deletion, and modification* of tuples in relational database systems, and *method executions* in OODBMS. The specification of a data manipulation event must include a site specification, if a corresponding data manipulation operation exists at different sites. A transaction event denotes the *begin*, the *commit* and the *abort* of a distributed transaction signaled by the *distributed transaction manager.* In addition, each *subtransaction* issues a *local_begin* and a *local_commit* or a *local_abort.* An abstract event denote an event which is triggered from outside a database system either by users or by application programs. Composite events are complex patterns of primitive events which are defined using an event algebra with well-defined semantics. Composite events are composed of primitive events and/or other composite events with event operators. Five event operators, *conjunction, disjunction, sequence, iteration*, and *negation,*

can be applied to events at local as well as remote sites. A specific event operator *concurrency* is applied to events only at remote sites. A time-stamp is applied to each event to indicate the time of the event occurrence. *Event parameters* capture the circumstances under which the event occurred, and are used to evaluate the condition and to execute the action of an ECA rule.

The semantics of primitive and composite events establishes when and where an event occurs and depends on the notion of physical time in distributed systems. A primitive event occurs and its timestamp is allocated when the event is detected. The semantics of a composite event depends on the timestamps of constituent primitive and composite events and event operators. Timestamps contain information on their original sites, and local and global representatives. The notions of time order relate to physical time and temporal order.

On event detection issue, the architecture and the algorithms for the detection of composite events at system runtime are developed. Event detectors are distributed to arbitrary sites and composite events are evaluated concurrently. Each site contains a *local event detector* (LED) and a *global event detector* (GED). Local event detectors detect local events and use centralized detection mechanisms. Detected local events are notified at local or global event detectors. Each global event detector must have registered their interest in a specified event type, and evaluates events received from multiple sites. Detected events are either signaled to the site's rule manager and/or are sent to registered global event detectors for further evaluation. In global event detection, global event trees represent global composite events. Nodes are labeled with event operators and leaves are labeled with event expressions. The root node is related to the outermost event operator of a global event expression. Occurrences of local events are signaled from corresponding local event detectors and occurrences of global events are signaled from corresponding global event detectors. Newly arrived events are injected into their corresponding leaves and flow upwards in the global event tree. Asynchronous and synchronous evaluations are considered when evaluate the nodes of global event trees. In synchronous evaluation, nodes are evaluated with respect to site failures and network delays, and events are forced to be evaluated in the system-wide order of their occurrence. Each node is evaluated regarding the 2gg-restricted temporal order of all events which may participate in an occurrence. The evaluation of a node blocks until all corresponding sites have been checked for relevant occurrences. This evaluation policy is suitable for applications requiring a high degree of consistency and reliability. Asynchronous evaluation is characterized by the ad hoc consumption of signaled event occurrences. Nodes are evaluated immediately on the arrival of suitable event occurrences when evaluate a global event tree. Delayed events are not considered. So, events are not evaluated in the order of their occurrence and an event detection is not blocked by delayed events. This evaluation policy is suitable for real-time applications, or any application that require fast response times. The prototype implementation realizes the algorithms for the detection of composite events with both asynchronous and synchronous evaluation. Primitive event occurrences are simulated by distributed event simulators. Several tests are performed illustrating the differences between asynchronous and synchronous evaluation.

3 Summary of Snoop and Snoop Preprocessor

ECA (event-condition-action) rules in active databases need to be complemented with an expressive event specification language. Event specification will significantly improve modeling of complex applications where temporal and external events are needed in addition to database events.

SNOOP [8, 9] is an event specification language used in Sentinel for specifying ECA rules. It defines event and rule specification, supports event operators and parameter context. spp is the preprocessor for SNOOP.

In a centralized active database system events (local event) occur at a single site, whereas in a distributed active database system event (global event) occurrences are related to many sites. The event specification language should reflect both local and global event characteristics. In this section, an overview of Snoop and its preprocessor are presented. Extensions to support global events in Snoop and spp is sketched.

3.1 Event Classification

An event is an atomic (happens completely or not at all) occurrence. A logical event specifies an event at the conceptual level, while a physical event is the point of detection of a logical event. Logical events are mapped to physical events uniquely, whereas a physical event may correspond to one or more logical events. Two event modifiers [10] , *begin-of* and *end-of,* are defined to map the occurrence of a logical event onto the physical level in a centralized database system.

Events in a distributed database system involve local events from different sites and are constructed based on these local events.

A hierarchy of event classes can be organized according to the structure and behavior. Four types of events can be identified in a distributed database system:

- Local Primitive Event

  Local Primitive Events are events that are predefined in that application using primitive event expressions and can be detected by a mechanism embedded in the system [11]. *Local Primitive Events* are classified into *database events* and *temporal events. Database events* refer to database operations to manipulate data, such as insert, update, delete and retrieve. *Database events* can be transformed into events using *event modifiers* (*begin-of* and *end-of*). *Temporal events* refer to specific points on the time line. An *absolute temporal event* is the event specified with an absolute value of time. It is defined in *(hh:mm:ss)/mm/dd/yy* format. A *relative* event is an event corresponding to a time point and is specified with a reference point and the offset. It has the format like event + [*time-string*], while *event* refers to any event allowed in SNOOP, and *[time-string]* refer to the time offset. External events are the events managed by un-local processes and signaled by applications. When an external event is signaled to the DBMS, the parameters are explicitly specified and supplied.

- Local Composite Event

  Local composite events are composed of *local primitive events* and other *local composite events* by applying event operators [12] . Since all the constituent events of a local composite event is defined locally, a centralized detection mechanism is used for local composite event detection.

- Global Primitive Event

  Global primitive events are events that are defined and detected outside of the current application and are referenced by the current application in a distributed database system. Any event (either local primitive or local composite) defined in any application can be a potential global primitive event. Since a global primitive event is defined outside of the application, the information needed by a local application which use this global primitive event should include the global primitive event name and the application name. Other information, like parameters of the global primitive event, are supplied to the local application when this global primitive event is signaled. The parameter computation is handled by the system and is transparent to the user application. According to the above deduction, two attributes are introduced in the global event specification: *App_name* and *Event_name*. They indicate that the event with the name *Event_name* is defined by application *App_name*.

- Global Composite Event

  Global composite events are related to event occurrences from many sites (including the local site). They are constructed with *local primitive events, local composite events, global primitive events* and other *global composite events* by applying subset of event operators defined in Snoop. At least one of the constituent event is a global event (primitive or composite) in a *global composite event expression.*

3.2 Event Operators

Event operators in Snoop are used to construct composite events. Ten event operators are supported in Snoop and are explained briefly in the following section.

- Conjunction: E_AND = e1 ^ e2

  The conjunction operator is applied if both events e1 and e2 occur. The order of two event occurrences is irrelevant.

- Disjunction: E_OR = e1 || e2

  The disjunction operator is used to denote the event occurrence when either e1 or e2 occurs.

- Sequence: E_SEQ = e1 >> e2

  The sequence operator denotes that event e1 happens before event e2. The time of occurrence of e1 has to be less than the time of occurrence of event e2.

- Negation: E_NOT = - (e1, e2, e3)

  The negation operator is applied when there is no event occurrence of e2 between the closed interval formed by e1 and e3.

- A: E_A = A (E1, E2, E3)

  The non-cumulative aperiodic operator is used to express the occurrence of an aperiodic event bounded by two arbitrary events. The event is signaled each time E2 occurs during the closed interval defined by the occurrences of E1 and E3. This event can occur zero or more times.

- A*: E_A* = A* (E1, E2, E3)

  The cumulative aperiodic operator denotes that event occurs only once when E3 occurs and accumulates the parameters for each occurrences of E2 in an interval formed by E1 and E3. It's useful to describe an event that occurs more than once in an time interval.

- P: E_P = P (E1, E2, E3)

  P is used to denote a periodic event that repeats itself within a constant finite amount of time. The event is signaled for each amount of time E2 in the interval (E1, E3].

- P*: E_P* = P (E1, E2, E3)

  P* is a cumulative variant of P and occurs only once when E3 occurs. The time of occurrences of the periodic event is accumulated whenever E2 occurs.

- PLUS: E_PLUS = E1 + [T]

  PLUS operator is used to denote events that occur when T time units are elapsed after E1 occurs.

3.3 Parameter Context

Four parameter context are currently supported in Sentinel.

- Recent

  In Recent context, only the most recent occurrence of the initiator for any event that has started the detection of that event is used, and all the events that can not be the initiators of the event are flushed. An initiator of an event will continue to initiate new event occurrences until a new initiator occurs.

- Chronicle

  In Chronicle context, the oldest initiator is paired with the oldest terminator for each event. Once occurrences of the constituent events are used, they cannot participate in any other occurrences of the composite event.

- Continuous

  In continuous context, each initiator of an event starts the detection of that event. A terminator event occurrence may detect one or more occurrences of the same event.

- Cumulative

  In cumulative context, for each constituent event, all occurrences of the event are accumulated until the composite event is detected.

3.4 SPP – Snoop Preprocessor

spp is the preprocessor for SNOOP language. First it preprocesses event and rule specifications defined by the user in Snoop, and inserts translated c++ code into the application program. Then it wraps all the methods of the reactive class defined in the user application. At the same time, it creates several files used by the rule editor and the global event detector.

In Sentinel, *spp* is integrated with OpenOODB preprocessor *ppCC* for wrapping methods of the reactive class and creating method signature files.

3.4.1 Snoop BNF

event_exp ::= E1

E1 ::= E1 **OR** E2 | E2

E2 ::= E2 **AND** E3 | E3

E3 ::= E3 **SEQ** E4 | E4

E4 ::= **Not** (E1, E1, E1)
| A (E1, E1, E1)
| *A** (E1, E1, E1)
| P (E1, [*time string*], E1)
| P (E1, [*time string*]:*paramater*, E1)
| *P** (E1, [*time string*], E1)
| *P** (E1, [*time string*]:*paramater*, E1)
| [*time string*]
| E1 PLUS [*time string*]
| (E1)
| *event_name*

event_name ::= *name*
| *Eventname:Objectname*
| *Eventname::AppId*

AppId ::= *Sitename__Appname*

name ::= Identifier

Eventname ::= Identifier

Objectname ::= Identifier

AppId ::= Identifier

3.4.2 Event and Rule Specification

SNOOP is an event specification language that supports several types of primitive events and event operators for constructing complex events in a distributed environment. In addition to the traditional database events, it supports external, temporal, global, and composite events.

Event specification refers to primitive event, global primitive event and composite event. The syntax of the event specification in SNOOP is as follows:

event_spec ::= **event** *event_modifier* method_signature

| **event** event_name = *event_exp*

| **begin** (event_name)

| **end** (event_name)

| **begin** (event_name) && **end** (event_name)

| **end** (event_name) && **begin** (event_name)

event_modifier ::= event_name

rule_spec ::= **rule** rule_name (event_name,

condition_function, action_function

[,[*parameter_context*],[*coupling_mode*]

,[*priority*],[*rule_trigger_mode*]])

parameter_context ::= RECENT | CHRONICLE | CONTINUOUS

| CUMULATIVE

coupling_mode::= IMMEDIATE| DEFERRED| DETACHED

priority::= positive integer

rule_trigger_mode::= NOW | PREVIOUS

In SNOOP, primitive events are specified using event_modifiers *begin* and *end*. The default event modifier of a primitive event is *end*. A global primitive event is defined with the format *event_name::application_name* which means that event *event_name* is defined and detected by application *application_name*. A composite event is defined using the event operators : AND, OR, SEQ, A, A*, P, P*, PLUS. SENTINEL supports both *class-level* and *instance-level* events. *Class-level* events are the events defined inside the class declaration. Each instance of this class can

trigger the events defined inside the class when corresponding methods are invoked. *Instance-level* events are the events related to a specific instance of the class and are defined outside of the class declaration. The class name is specified in the event expression.

4 Architecture

This section discusses two architectural alternatives for the global event detector. Four design approaches are presented and compared with each other.

4.1 Introduction

In a distributed computing system, event detection should monitor the behavior of events in a distributed environment. This requires a mechanism to detect events occurring ont only at a local site, but also at other remote sites. Global event detector is responsible for detecting events of inter-applications in a distributed database environment. It recognizes the occurrence of events, collects and records their parameters, and passes it to application rule managers to trigger the action of ECA rules.

Since global event detection involves multiple applications from different sites, the special features of distributed systems increase complexity of event detection as compared to centralized systems. These special characteristics of distributed systems include concurrent processes running at multiple autonomous sites, lack of global time, message delays between sites, etc. Since the communication between applications plays a significant role in system performance, the architecture we choose for global event detector should reduces the communication overhead as much as possible.

4.2 Architecture Alternatives

There are two approaches to detect global events in a distributed environment.

1. Distribute global event detection among applications

   In this approach, all events (local or global) are detected in local sites, and applications communicate with each other directly to exchange messages. Every application has a local event detector and a global event detector to detect events. Global event detector in this case plays the role of a message sending/receiving than an event detector. Local event detector detects all the events defined either in the application it resides on or in other application from any site. After each event is detected locally, it will send event notification and parameter list to the global event detector to transfer messages to other applications as necessary.

   Since each application has to communicate with every other application directly, message exchange overhead is significant in this case. The system performance will decrease with increase in the number of applications involved in global event detection. Yet from a designer's point of view, this approach makes implementation much easier, since the main task of global event detector is to transfer messages between applications.

   The architecture of this alternative is shown in Figure 1.

   In this approach, LED and GED are part of each application. All events are detected locally, and global events are notified by GED through network communication.

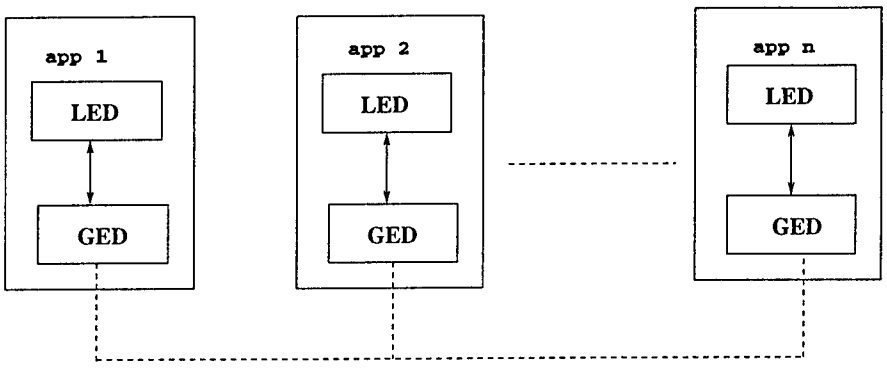


Figure 1: Architecture of a Distributed Event Detector System

2. client/server architecture

   This approach uses the client/server model to centralize global event detection on a server site. Global event detector is a separate server process. It receives requests and messages from clients, builds its global event graph where global events are detected, and sends the event notification to clients when each event is detected. Each application runs LED to detect local events, and communicates with other applications through GED.

   Client/server architecture centralizes the global event detection on a server site, thereby decreasing the communication overhead between applications, and is likely to increase system performance. On the other hand, since global event detector needs to detect global events in addition to transfer messages between applications, the implementation is more complicated. In Sentinel, client/server model is chosen to implement the global event detector (GED).

   The architecture of client/server approach is shown in Figure 2.

   In the above graph, GED is running as a daemon on the server site. Each client has a LED to detect local events. LED communicates with GED using the remote procedure calls (RPC).

4.3 Asynchronous Communication Between Client and Server

To detect a global event, each client should send global event detection request and event specifications to the server, and receive the events to be notified from the server. This sending and receiving action should be carried out in an asynchronous manner. That is, after the client sends requests to the server, it will not be blocked waiting for reply. The client application should be able to continue its work and receive the notification from the server meanwhile.

Four approaches are discussed below to meet this requirement.

- Heavyweight Process

  This approach is shown in Figure 3.

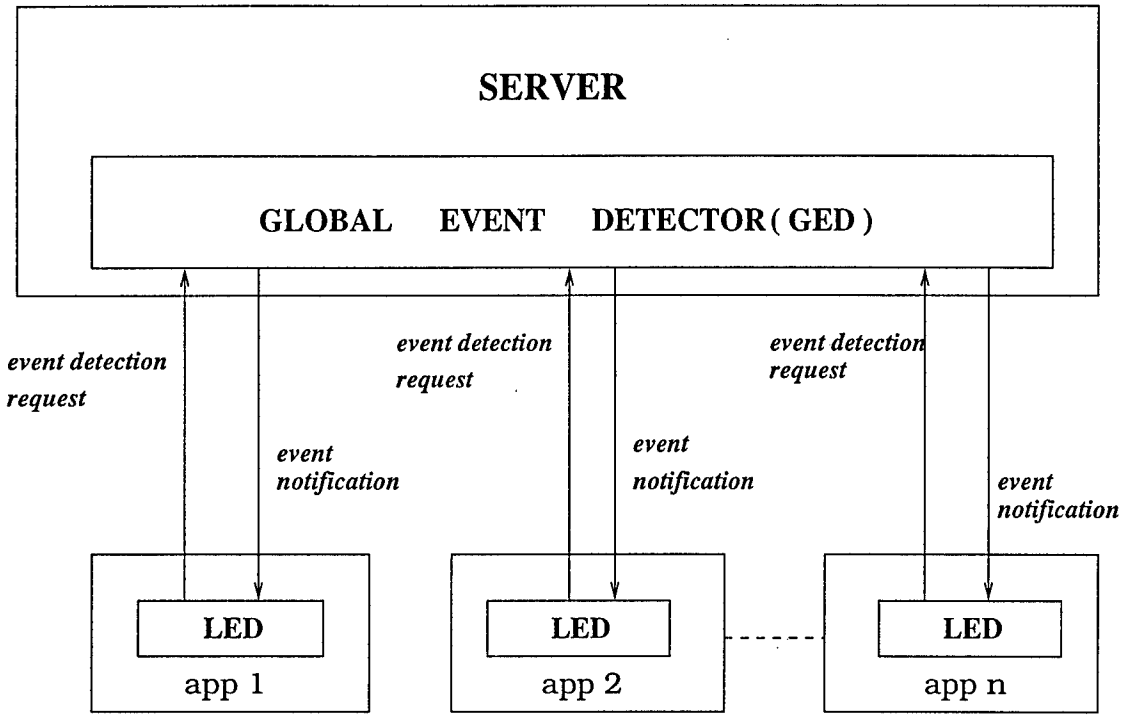


Figure 2: A Client-Server Architecture of Global Event Detector

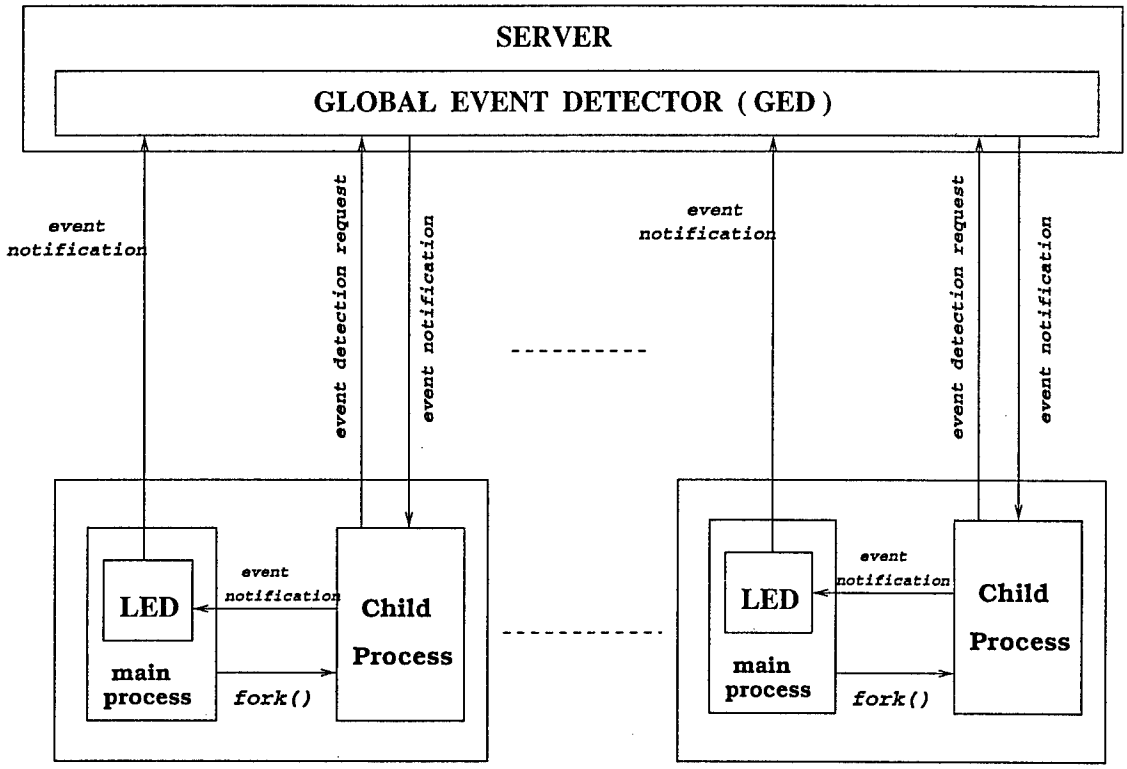


Figure 3: A Heavyweight Process Design Model of GED

Each client forks a child process to communicate with the server. The communication includes sending global event detection requests and receiving notification back from the server. The parent process detects local events and processes user application. This approach separates the communication task and event detection task into two processes, and these two tasks execute independently without affecting each other.

The main problem with this approach is that the memory space used by child and parent processes are different. Each process uses its own memory space, and thus the variables used by one process are not visible to the others. This will create data inconsistency if the two processes try to process and evaluate using the common data structures.

To detect a global events, child and parent processes need to access and update global variables. For example, an event name table is maintained in the client site. This event name table records all the local event names that are needed to send the notification to the server after they are fired. The table is created upon the messages sent from the server in the child process, and are accessed by LED when each local event is detected in the parent process. Since the table is updated dynamically during run time, the updates to this data will not be visible to the parent process. One way to solve this problem is to use IPC tools to transfer messages between two processes. Message queue, pipe... are alternatives for inter process communication. This will increase implementation complexity and decrease system performance without any gain.

- Asynchronous I/O

  An interrupt driven socket I/O method is used in this approach. This signal-driven I/O allows the process to notify the kernel when a specific descriptor (socket) is ready for I/O.

  In this implementation, the server gets a client socket address after the client make a RPC request. It also get its own socket ID from the transport handler. When a global event is detected by the server, it will send a signal to the corresponding client with this client socket address obtained earlier. When the client is notified (interrupted) by this signal, it will jump to a signal handler to continue its communication with server.

  This asynchronous I/O approach is very straightforward, and simplifies the implementation. The main problem is the signal security. The signals sent from server may be lost (will not be caught by the client) if they are not implemented in a proper way.

  The diagram for this approach is shown in Figure 4.

- Two way Remote Procedure Call (RPC):

  In this approach, the role of client and server is not clearly differentiated. Both the client and the server site can act as a client and a server. Client makes RPC calls to the server

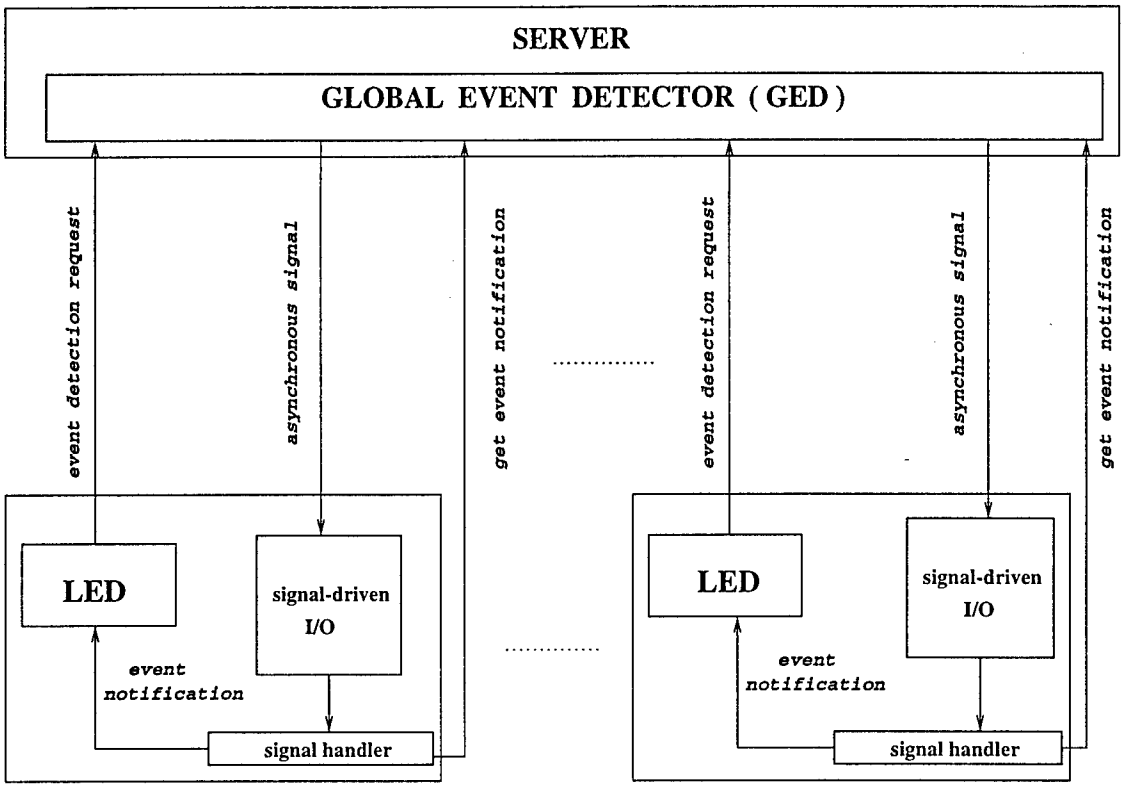


Figure 4: An Asynchronous I/O Design Model of GED

to send global event detection request, and receives RPC calls from server to receive event notifications at a later time.

So, by adding a second request-reply step, server makes RPC call back to the client process whenever an event is detected. Client provides services in its own procedure to catch response sent from the server. This method looks clear and simple, yet creates problems when more than one application is running on the same machine. Since all client applications provides the same services to the server, they have the same program number, version number and procedure number. When the server sends a response back to the client, it will go to a specific client according to the client ID:*(host name, service program number, version number, procedure number)*. Since several client applications running at the same machine will have the same client ID, only one of the client process (the first one who has registered the RPC services) will catch the RPC request from the server. This approach only works in situations that allow only one application to be running on one machine, and will not meet the requirement when several applications are running on the same machine.

The diagram is shown in Figure 5.

- combination of RPC and socket

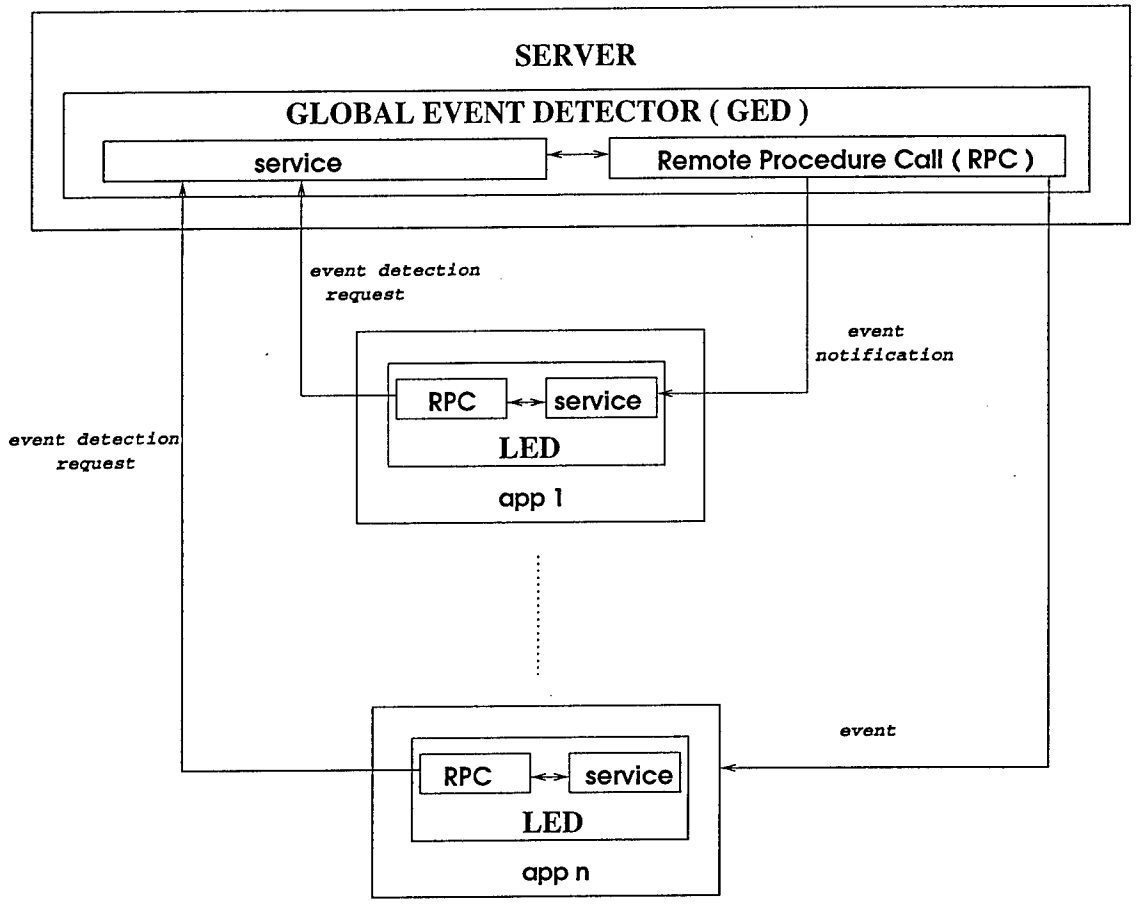


Figure 5.5 A Two Way RPC Design Model of GED

Figure 5: A two way RPC Design Model of GED

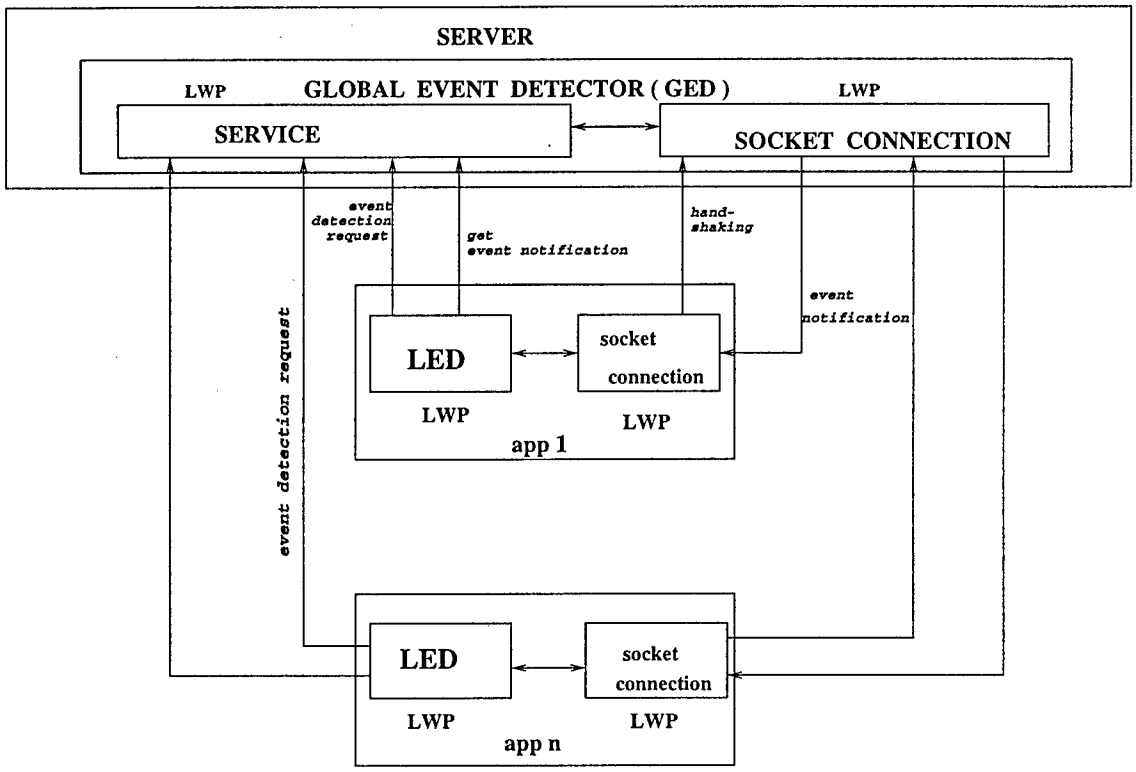


Figure 6: A combination of RPC and socket Design Model of GED

This approach is an improvement of approach 3 (*two way RPC*) discussed above. In order to make the server communicate with different client processes running on the same machine, a socket communication interface is added to meet this request. Since each process has its own socket ID, the server can send response back to a specific client according to this socket ID no matter where the client process is running. When the client receives a message from the server through its socket interface, it will make RPC calls to the server to receive event notifications.

The design is as follows: A client process makes a socket connection with server during its hand-shake with the server, and the server records the socket ID of this client at the same time. Whenever an event is detected by the server, according to its event subscribers (client application ID), the server will send a response to each such subscriber through socket according to the subscriber socket ID. After the subscriber (a client process) has received this response, it will make RPC calls to the server to get the event notification (event name and parameter list). In Sentinel, this approach is used to implement global event detector.

The architecture is shown in Figure 6.

5 Extensions to Snoop for Specifying Global Events

To support ECA rules in a distributed environment, Snoop – the event specification language of Sentinel has been extended to specify global events for event detection. This section introduces the global event specification in Snoop, and presents the details of preprocessing.

5.1 Global Event Type

Two global event types are supported in Sentinel: global primitive event and global composite event.

Global primitive event is an event that is defined and detected outside of user application. Any event, including primitive and global events defined by any other application can be defined as a global primitive event in the current application. *Event name, application name,* and *host name* are used to compose this global event type specification, and are the only information that needs to be provided by user applications for this global primitive event detection. The event detection and network communication details are implemented by the *global event detector* and are transparent to the user.

Global composite event is an event that is composed by event operators and at least one of its constituent event should be a global event. Any event, including primitive events, composite events and global events can be a constituent of a global composite event. A global composite event can be detected either at the local site or at the GED. The site of the global composite event detection is determined at the time of preprocessing events and depends on the global composite event specification. Each global composite event specification corresponds to an event tree constructed by the event detector. The details of site selection and event detection are discussed in the following section.

5.2 Global Event Specification

5.2.1 Global Primitive Event

The global primitive event specification is as follows:

local_event_name = *remote_event_name*::*host_name*__*application_name*

local_event_name is the event name defined by user's application. It can be used as the constituent event name of composite events. *remote_event_name* is the event name that is defined in other application where this event is detected. *host_name* and *application_name* denote the name of the machine and application where this remote event is specified. It is assumed that an event *remote_event_name* has been defined and will be detected by the application *application_name* on the machine *host_name*. (If a global event name is misspelled, it will be treated as if that event will never get detected.) This global primitive event can either be specified as a stand-alone event definition, as shown above, or it can be used as a constituent of a composite event definition.

5.2.2 Global Composite Event

The global composite event specification is similar to local composite event specification except that at least one of the constituent event must be a global event. This constituent event can be presented as a global event name or a global primitive event specification.

Below, we present some examples of global event definitions in Snoop:

```
class STOCK: public REACTIVE  {
    public:
    STOCK();
    int get_total_stock();

    event end(e1) int buy_stock(int qty);
    event begin(e2) int sell_stock(int qty);

    event g1 = STOCK_e2::sugar__app1;
    event g2 = e_IBM::manatee__app2;
    event g3 = (g2 >> STOCK_e1::eagle__app3) ^ g1;
    event g4 = !(g1, g2, g3);

    rule gr1[g1, cond1, test_action1, RECENT];
    rule gr2[g3, cond2, test_action2, RECENT];
    rule gr3[g4, cond3, test_action3, RECENT];
```

Event *e1* is a local primitive event which occurs after the method *buy_stock* is executed. Event *e2* is a local primitive event that is triggered before the method *sell_stock* is executed. Event *g1* is a global primitive event that is triggered when the event *STOCK_e2* is detected by application *app1* on *sugar* site. Event *g2* is a global primitive event that raises when the event *e_IBM* is signaled by application *app2* on site *manatee*. Event *g3* is a global composite event specified by operator (>>) and (^). Its constituent events are composed with global composite event *g1*, global primitive event name *g2*, and global primitive event specification: *STOCK_e1::eagle__app3*. Event *g4* denotes a global composite event with "!" operator. Three rules *gr1, gr2, gr3* are defined for events *g1, g3, g4*.

Since the above events are declared at the *class* level, they are detected for each instance of class **STOCK**.

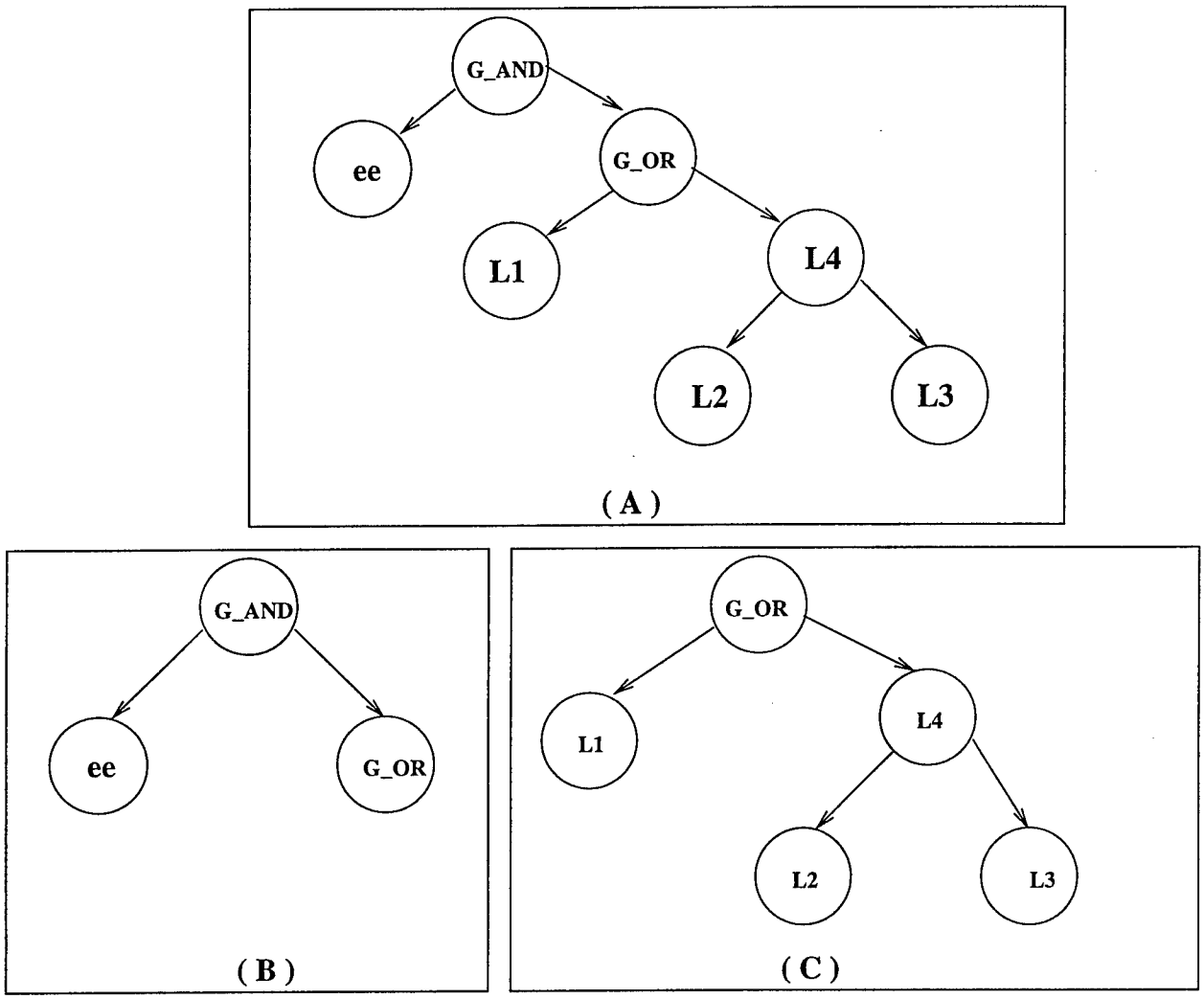


Figure 7: A Global event tree (1)

5.3 Alternatives For Global Event Detection

In Sentinel, global primitive events are detected by the corresponding remote sites, and the event notifications are sent to the GED if the event is used by other applications. Unlike a global primitive event, the global composite event detection is more complicated since the constituent events can be either local or global. A composite event can be detected either at the local site or by the GED server. The appropriate choice of global event detecting site plays a significant role in system performance since it involves network communication. Based on the event detection site, two alternatives are discussed, and the following examples (shown in Figure 7(A)) is used to compare these two approaches.

In the example, a global event *G_AND* is composed by a global composite event *G_OR* and a local event *ee*. Event *G_OR* is defined on a remote site and is composed of four local events: (*L1, L2, L3, L4*) with *OR* operator.

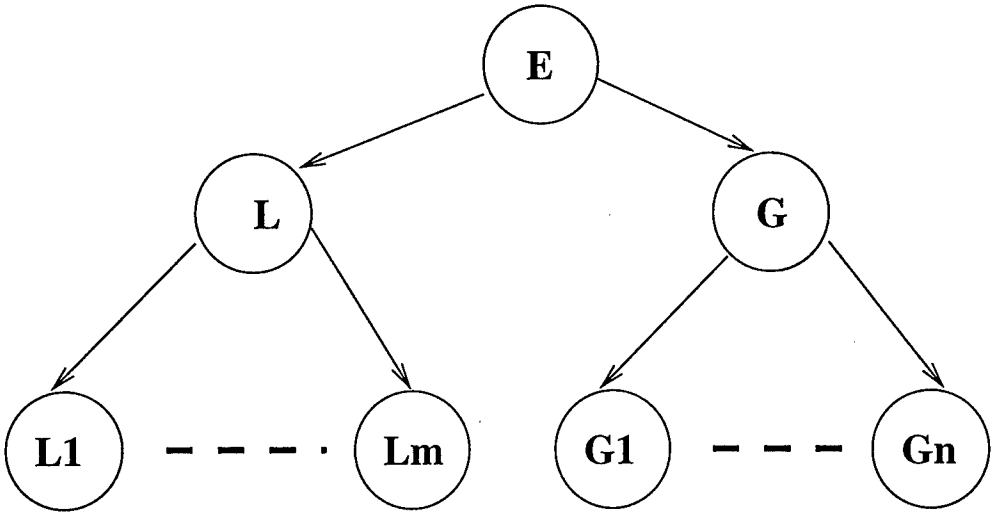


Figure 8: A Global event tree (2)

1. Global composite event is detected at the GED

   In this case, the whole global composite event tree is sent to the GED server. Every leaf node of the event tree is to be detected by its corresponding site and the notification will be sent from this site to the GED after this event is detected. For the above example, the number of communications between node sites and the server is "4", since the event tree has four leaf nodes (corresponding to primitive events). When a global composite event (the root node of the event tree) is signaled by the GED, an event notification will be sent from the server to the corresponding local site where this event tree is sent from. So, the total number of communication between clients and the server is: $4 + 1 = 5$.

   To consider the general case, we introduce a global composite event tree which is shown in figure 8.

   G is a constituent event of global event E and is defined by n local events $G1$ ... Gn on a remote site. $L1...Lm$ are other constituent events of event E. Since the whole event tree is detected by the GED, each leaf node need to send notification to the server when the corresponding event is detected. The number of messages between clients and the server is: $m + n + 1$.

2. Global composite event is detected at the local site

   In this case, all the constituent local events of a composite event are detected on its corresponding remote sites. Only when the global composite event is detected, the event notification is sent from the remote site to the GED. In the above example, the event subtree for node *G_OR* (as shown in Figure 5.1(C)) is detected on the remote site. The event tree sent to the GED will be like the one shown in Figure 5.1 (B). Since only the node *ee* and the node

G_OR need to send the notification to the GED, the number of communications will be 2 + 1 = 3.

To consider the above general case, the event tree of the constituent event G is detected on the remote site, the number of messages between clients and the server is: $m + 1 + 1 = m + 2$.

In Sentinel, alternative 2 is used for global composite event detection. It reduces the network communication overhead and thus improve the system performance.

5.4 Implementation Detail

Global events are related to the event detection from many sites, and involves communication between client and server. Snoop preprocessor *spp* generates necessary c++ code for each application, and a global event specification file for sharing information between client and the server. The only thing any application needs to do is to specify the global events according to the SNOOP syntax. The implementation details are transparent to the user's application. Since composite events are detected by an event tree (discussed in section 4), snoop preprocessor *spp* need to create corresponding messages according to the event definition. In *spp*, a *remote* node is generated for a certain global event and is presented as c++ code which is inserted into user's application source code. Also, a global event specification file is created for each application. This file will be read and converted to an event detection graph during run time. This graph will be sent to the GED server for constructing the global event detection graph on the server site.

Following are C++ codes generated by *spp* for global event specifications in the above example:

```
REMOTE *STOCK_g1 = new REMOTE("sugar__app1", "STOCK_g1");
REMOTE *STOCK_g2 = new REMOTE("manatee__app2", "STOCK_g2");
REMOTE *STOCK_g3 = new REMOTE("REMOTE", "G_comp_1");
REMOTE *STOCK_g4 = new REMOTE("REMOTE", "G_comp_2");
```

The global event specification file is generated by *spp* as:

```
0 STOCK_g1 1 * global sugar__app1 STOCK_e2 *
0 STOCK_g2 0 * global manatee__app2 e_IBM *
0 tmp0 0 * * eagle__app3 STOCK_e1 *
2 tmp1 0 * tmp STOCK_g2 tmp0 *
1 STOCK_g3 1 * G_comp_1 tmp1 STOCK_g1 *
8 STOCK_g4 1 * G_comp_2 STOCK_g1 STOCK_g2 STOCK_g
```

5.4.1 REMOTE Object

For global event detection, a *REMOTE* class is added to the *REACTIVE* class hierarchy (the details are discussed in section 5). Each *REMOTE* object corresponds to a global event that will be detected and notified by the GED. In a *REMOTE* class object, the parameters of the class constructor are the application ID with the *SiteName__ApplicationName* format, and an global event name created by the server. For example, in the REMOTE instance *STOCK_g1*, the application ID is *sugar__app1*, and the event name is *STOCK__g1*.

Since global event detection involves events from many sites, the server may receive events with the same name from different sites. To avoid duplicate event names from different sites, a unique event name should be given to each global event sent to the server. In *spp*, we rename the global event name according to its event type. For a global primitive event, the unique event name is composed by the following format: *G_comp_eventnumber. eventnumber* is a sequential number for each global event sent to the server. It is assigned to global primitive events in the order in which they are received by the server. Both the client site and the server site keeps this unique event name for event detection. When a global event is notified to the server, it is sent to the corresponding client site and notifies the local event using this unique name. Each such object becomes a leaf node in the global event detection graph. The details of the event graph are discussed in the following section.

5.4.2 Global Event Specification File

A global event specification file is created for each application. This file contains the information about global events that are detected by the GED. An event graph is created using this file at run time, and is sent to the GED for constructing the global event graph on the server site for global event detection. The event graph sent from local site to the server is composed of event detection trees. Each tree is related to a global event. A global primitive event is a special tree which has only one node.

There are two kinds of global primitive events declared in user applications. One is a stand-alone event definition that will be used by composite events and rule definitions. In the above example, global primitive event *g1* is declared and is used by rule definition *gr1*. This stand-alone global event is related to a REMOTE class object (*STOCK_g1* in the above example) in local application and is recorded into the global event specification file with an unique name *classname_eventname* (*STOCK__g1* in the above example). The other kind of global primitive event is the one that is a component event of a composite event and is defined within the composite event definition. For example, in the composite event definition: *event g3 = (g2 >> STOCK_e1::eagle__app3 ^ e1)* the global primitive event *STOCK_e1::eagle__app3* is a constituent event of the global composite event *g3*. Since *g3* is detected at the server site, this global primitive event notification only needs to be sent to the server by the application *eagle__app3*. The local application will not receive this event

notification. Thus no REMOTE class instance is created for it. In the event specification file, this global primitive event is recorded using an unique event name assigned by *spp* with the format **tmp***eventnum* (*tmp0* in the above example). *eventnum* is the unique sequential number created for this event. For each global composite event, *spp* creates a REMOTE instance in the application and records this event into the event specification file.

The event specification file is used to create an event graph which will be sent to the server by the user application. Since the GED server builds its event graph according to this file, this file records all the information that is needed to detect global events.

The format of the global event file is:

operator EventName sendback_flag AppID CompEventName
constituent_eventname1 constituent_eventname2 constituent_eventname3

operator is an unique number related to an event operator. For example, "1" denotes operator AND, "2" denotes operator SEQ, "0" denotes a global primitive event, and "8" corresponds to NOT operator.

EventName is the unique name of a global event assigned by *spp*. For example, *STOCK_g1* is the name for global event *g1*, and *tmp0* is the name for global event *STOCK_e1::eagle__app3*.

sendback_flag is the flag which determines whether the notification of this global event will be sent back to this site from server. The value of the flag is *1* if the global event notification is to be sent back to this local site after it is fired, otherwise the value is *0*. Since some global event, like the global primitive event *tmp0* in the above example, is just a constituent part of a composite event tree, it is not necessary to send it back after it is fired in server. The corresponding *sendback_flag* is *0*.

AppID is the ID of the current application that generates this event specification file. Since the application ID is composed with *SiteName__AppName*, *spp* cannot get it during the compilation time. This field will be filled as * temporarily by *spp* at this time. Later in the run time the value of this field is assigned by the GED interface of the local application. For example, *eagle__Demo* will be assigned for the application *Demo* running on *eagle*. *AppID* is used by the GED to connect to an application when certain global events are detected and are sent back to this specific application.

CompEventName is the name of a global composite event that is used by the GED. It is created with the format **G_comp_** *EvntNum* (*EvntNum* is a sequential number assigned by the server). Since this field has no meaning for a global primitive event, it will be assigned a * or *tmp*. In the above example, *G_comp_1* is created for global composite event *STOCK_g3* and *tmp* is used for global primitive event *tmp1*.

constituent_eventname1,*constituent_eventname2* and *constituent_eventname3* are used to present constituent event names for a composite event. Since the maximum number of the constituent event in a composite event is 3, three fields are used for this purpose. The default value of these

fields are *. For the global primitive event, *constituent_eventname1* and *constituent_eventname2* are used to denote the *application ID* and the *event name* that this global primitive event is composed of. In the above example, *tmp1* and *STOCK_g1* are two constituent events of the global composite event *STOCK_g3*. *STOCK_g1, STOCK_g2, STOCK_g1* are three constituent events of the global composite event *STOCK_g4*. For event *STOCK_g1*, corresponding values of these two fields are *sugar__app1* and *STOCK_e2* which means that the global primitive event *STOCK_g1* will be triggered when event *STOCK_e2* is detected by application *app1* running on site *sugar*.

Since GED builds the global event graph according to the event specification file, each client needs to send the contents of this file to the server at the beginning of the execution. There are two ways to transfer the message from a client to the server. One is to send the path and file name of this global event description file to server. The other way is to read the file in local site, create the corresponding linked list which contains the global event description, and send this linked list to the server. The implementation of the first case is simple and easy, yet it will cause problems for the server to access the file when the client and server are running in different file system. The second way is much more complicated and difficult to implement, however it is applicable in any file system. In SENTINEL, we have implemented the second alternative. Then this client/server model is applicable to an open system environment.

5.4.3 Flags In SPP

Several flags are provided by *spp* to facililate the use of global event detections and rule executions.

- **-s:**

  Since *spp* is integrated with OpenOODB preprocessor *ppCC*, the **-s** flag indicates to the *ppCC* to invoke *spp*. Any application that uses SNOOP language should use this flag.

- **-gen, -use:**

  In *spp*, an *event definition file* is created for each application. This file contains all the event and rule definitions in c++ executable format. This c++ code is inserted into the main module of an application by spp. Since the main module of an application can exist under a different path from other modules, it is difficult for the main program to get the event and rule definition file created from other modules. We use **-gen** and **-use** flags to solve this problem. **-gen** *filename* provides *ppCC* with the file name that contains the event and rule definition. **-use** *filename* provides *ppCC* with the path and file name that is created by the *-gen* flag.

- **-host, -port, -sg:**

  These three flags specify the *host name*, *port number* and *storage group* of the application. They are used by OpenOODB to define an OODB object instance to name the method

signature file as *HOST_PORT_SG.signature* and static event file as *HOST_PORT_SG.static*. Since the *host name, port number* and *storage group* number are unique for any application, the name of the signature and static files are unique.

- **-GS, -GD, -GF**:

  These three flags are for global event specification file. Global event specification file is the file that contains the global event information for the server to build the global event graph. Every local application that contains global event definitions will generate this file. Since several stand-alone modules in one application can contain their own global event definitions, each module can have its own global event specification file. To integrate these global event files from different modules into one, we need to use some flags. **-GF** *filename* defines the global event specification file name that is generated by the application. This flag is used for each module that contains global event definition using SNOOP language. **-GS** *filename* provides the path and name of the event specification file generated by **-GF** flag. **-GD** *filename* defines the final global event specification file used by the application. This file is used by the main module to create the linked list of global event information before sending it to the server to build the event graph.

5.4.4 Integrating SPP With ppCC

ppCC is the preprocessor of OpenOODB. It preprocesses class definitions and extends them with member functions for use by the OpenOODB system. In SENTINEL, *spp* is integrated with *ppCC* to preprocess SNOOP language. The sequence of executions of integrated *ppCC* is as follows:

First, *ppCC* calls the c++ preprocessor (*CC* with **-E** flag) for macro expansion. Then it calls *spp* if a **-s** flag is specified. After that, OpenOODB interpreter *ccpp* is invoked to extend class definitions, and the output file is sent to the c++ compiler *CC* to create the final object file.

6 Implementation of Global Event Detector

In section 4, we analyzed the architecture alternatives, and chose the client/server model to implement the global event detector. In this section, an overview of the local event detector is briefly introduced before describing the GED in detail. This includes global event detection requirements, extensions to the Local Event Detector (LED) for supporting global event detection, and implementation details of GED.

6.1 Local Event Detector

LED (Local Event Detector) is linked with an application for detecting local events. The architecture of LED is illustrated in Figure 9

An event graph is used for local event detection. The REACTIVE class is the class that contains procedures for dealing with the event and rule specification. As we mentioned before, every method of a REACTIVE class is a potential primitive event. EVNT_LIST is a linked list of EVNT_NODE. Every EVNT_NODE corresponds to a unique REACTIVE object. EVNT_NODE has a *begin_of* event list and a *end_of* event list. The *begin_of* list contains methods defined inside its corresponding REACTIVE class that will raise a primitive event before this method is executed. The *end_of* list records those methods that will raise a primitive event after this method is processed. Each node of the two event list points to a primitive event, that is, a leaf node of the event graph, from which composite events are constructed from. Each node of the event graph has an event subscriber and a rule subscriber which record the related rules and composite events. Whenever a primitive event is raised, it will notify its subscribers, which are their parent nodes. The parent nodes (composite events) will maintain the occurrence of its constituent event occurrences as part of their parameter lists which are stored separately for each context relevant to the node. If the composite event occurs by the current notification, it is detected and notified to its subscribers. The parameter list is recomputed to include the new occurrences. For details of the LED, refer to [13].

6.2 Global Event Detection Requirements

6.2.1 Distribution Of Event Detection

Since global event detection involves events from many sites, distributing event detection improves reliability and reduces message passing overhead. In SENTINEL, a LED (Local Event Detector) is part of each site, and a GED (Global Event Detector) is installed on the server. According to the global event specification and detection algorithms, some global events are detected on the client sites whereas others are detected by the server. Sharing the event detection between client and server decreases the communication overhead and event detection burden.

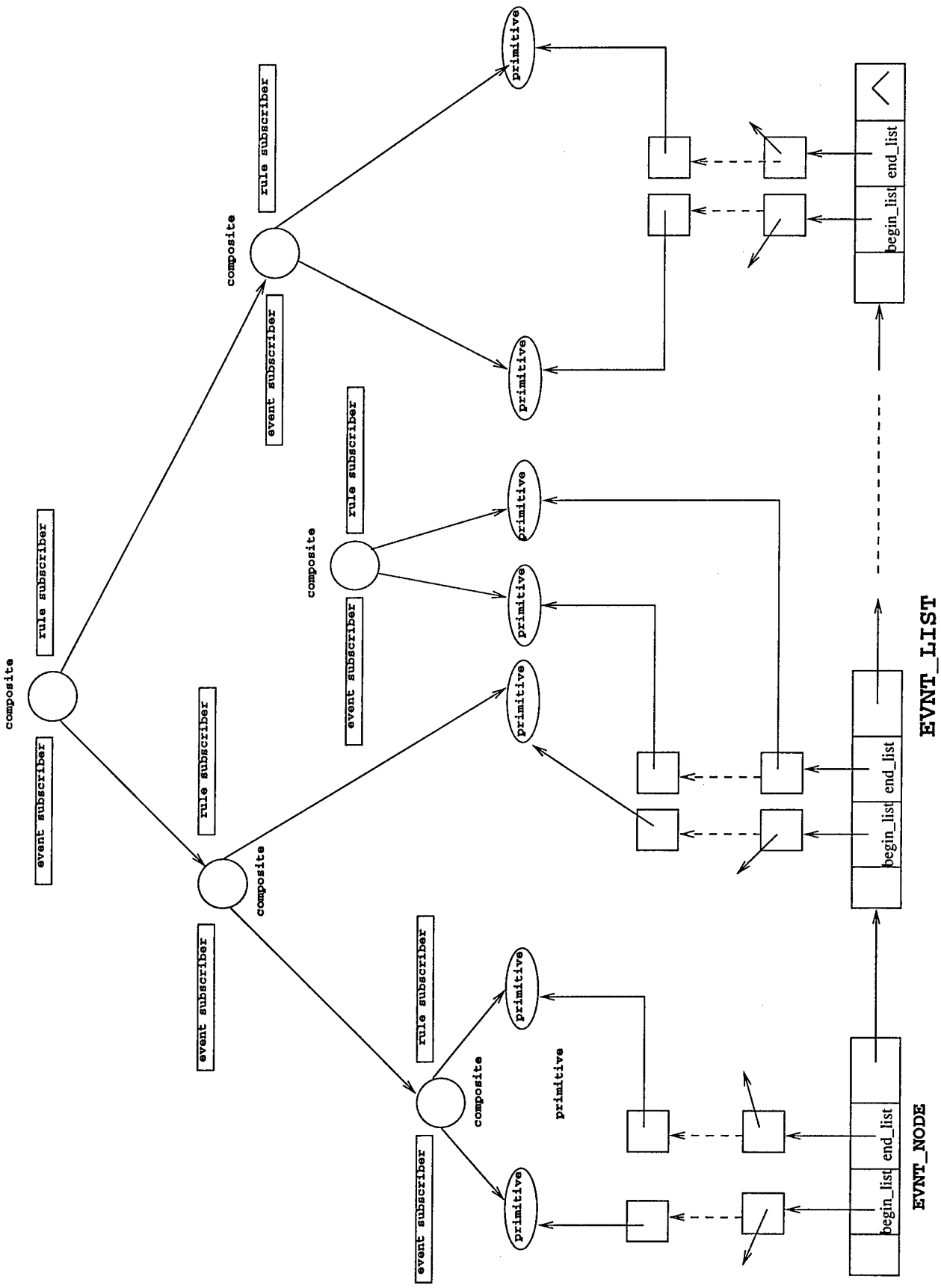


Figure 9: Architecture of Local Event Detector (LED)

6.2.2 Provide Event Detection Functionality To The User Application

Global event detection is the only task performed at the server site. Server accepts messages from clients, invokes the corresponding service, and sends the message back to clients. This allows the processes on the server to run as either a foreground or the a background job.

A Local event detector at the client site is integrated into the local application. It is implemented as a static library and provides the function calls for event detection. Since LED needs to communicate with the GED in the server, it should be running as a local daemon. Moreover, as an LED needs to communicate with local application to exchange messages, both LED and the local process should share the same address space. Forking a subprocess for LED does not meet this requirement. A light weight process (or a thread) is a better alternative to accommodate this goal. Multiple threads execute as concurrent execution streams sharing the same address space performing tasks associated with the desired services.

6.2.3 Global Event Detection Site

Since the communication overhead between client and server has a significant effect in system performance, it is very important to decide the place where global events are detected. Two approaches are discussed here.

- All global composite events are detected in local site.

  In this approach, constituent global primitive event of a global composite event are detected outside of the local application and are notified by the server. For each such global primitive event, the server needs to send an event detection request to its corresponding application, receives the event notification when the event is detected, and finally notifies to this local process. Since every global primitive event has to go through this process, the communication overhead is significant in this approach.

- Client and server share global composite event detection

  In this approach, global composite events are detected by either client application or the server. The place where a global composite event is detected is determined by the event expression. When a global composite event is to be detected by the GED, an event tree of this composite event is sent to the server by the client application when a client communicates with the server for the first time, and is to be used to update the event graph in the server site for global event detection. Only when a global event is detected by the server will the server send the notification back to the corresponding client applications. To avoid detecting and notifying each global primitive event by the server, this approach decreases the communication overhead substantially.

6.3 Extensions To Local Event Detector

Each client has a local event detector (LED) which is composed of an event detection graph. In addition to detecting local events, LED sends the local event notification to the server whenever this event is raised, and receives the event notification from the server when a global event is detected by the server. To accommodate the above requirement, LED is modified to include extensions to event class hierarchy.

6.3.1 Extension Of Event Class Hierarchy

To accommodate global events, a REMOTE class is added to the class hierarchy in LED. REMOTE class is a subclass of the EVENT class, and represents global event objects. Each global event that is detected outside of the application and notified from the server is an object instance of REMOTE class. There are two attributes of this REMOTE class: *App_ID* and *instance_number*. According to the place where a global event is detected, the value of *App_ID* attribute is assigned in a different way. If a global event is detected outside of the application process, *App_ID* is the ID of an application where this global event is defined and detected. An application ID is composed in the form of *SiteName__AppName* which denotes application *AppName* running on the machine *SiteName*. If a global event is detected inside the application, the value of *App_ID* is a reserved word *REMOTE*, which is to differentiate the global event instance from local event instance. The *App_ID* attribute combined with the *event_name* (an attribute of NOTIFIABLE class) make a unique ID for each global event. *instance number* is the occurrence number of this global event instance.

The extended class hierarchy is illustrated in Figure 10

6.3.2 Extended Local Event Detector (ELED)

The architecture of ELED is illustrated in Figure 11

Extended Local Event Detector (ELED) is an extension to LED to detect global composite events in the local site. Similar to LED, ELED is an instance of EVNT_LIST class that records information of all the global event instances. Each node of the EVNT_LIST linked list is related to a unique application and contains all the global event instances that are detected outside of this application. An event linked list which is an ELIST class instance is related to each node and contains all of the global event instances information that belongs to this specific application. Each node of such ELIST instance corresponds to a REMOTE node and become a leaf node of the event graph. A composite event can be constructed from REMOTE nodes, Primitive Event nodes, and other composite event nodes. Whenever a global event is detected outside of the application, a GED Interface will receive the event notification along with application ID and event parameter list from the server and further notifies ELED. ELED then determines the specific EVNT_LIST node according to the application ID and propagates the event notification to its corresponding REMOTE

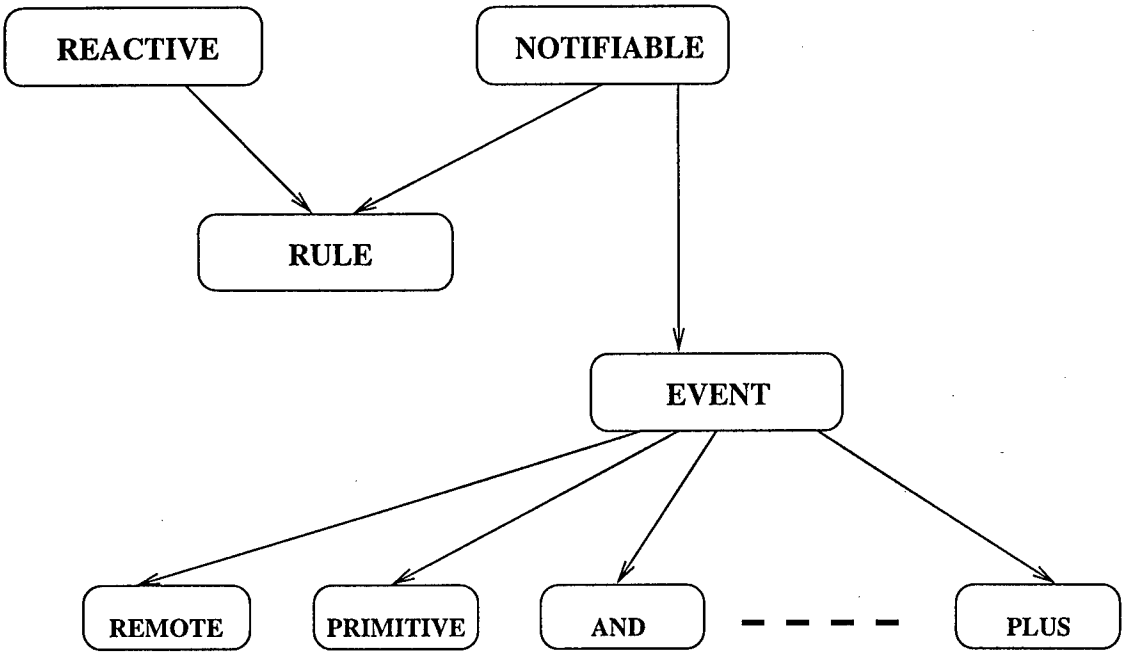


Figure 10: Event Class Hierarchy of LED

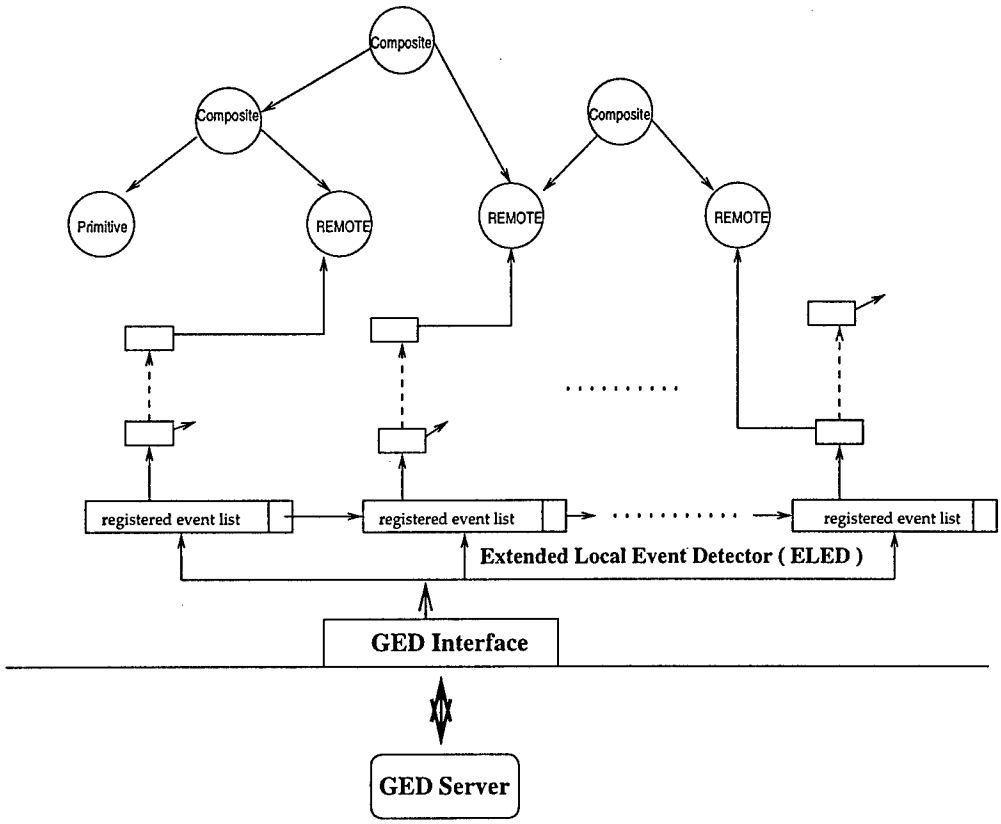


Figure 11: Architecture of Extended Local Event Detector (ELED)

event instance. According to its event subscribers and rule subscribers, a notified REMOTE event instance will further notify related composite events, that is, its parent nodes.

6.3.3 GED Interface

To extend LED for supporting global event detection, a network interface is needed to exchange messages between the client application and the GED. A GED Interface on the client site is implemented to communicate with the server. It generates an event tree list which contains global composite events that are to be detected by the server, sends this event tree list to the server, and receives the global event notifications from the server. In order to support different applications running on the same machine, a socket connection is built between client and server, and the unique socket address is recorded by the server for later message reply. In addition to the socket connection, a client application makes remote procedure calls to the server to request global event detection and receive event notifications from the GED.

6.3.4 Event Tree Propagation By Client

To decrease the communication overhead between a client and the server, we should send the composite event tree only if the global event nodes inside this tree are other than local event nodes. Since the composite event detection tree is generated according to the SNOOP operator semantics, one parent node can have at most three children. As a result, a sub-tree with less than or equal to two children is sent to the server only when at least one of the child event is to be detected by the server, a sub_tree with less than or equal to three children is sent to the server only when at least two of the children are to be detected at the server. According to the event tree sent from a client, the server builds the global event graph. The sub event trees sent to the server is created by *spp* when it parses and analyzes the global event definition in an application. A global event specification file that contains the global event tree information is created by spp. Global event trees are created from the global event specification file at run time and are sent to the server during the first handshake between a client and the server.

6.4 Implementation of GED

6.4.1 Client/Server Model

GED (Global Event Detector) is implemented using client/server module as illustrated in Figure 12

GED is installed on server and provides services to the clients by detecting global events. When a client sends requests to the server, the server sends the requirements to the local service, processes it, and sends the necessary information back to the client. In this way, clients communicate with each other through server in a transparent way.

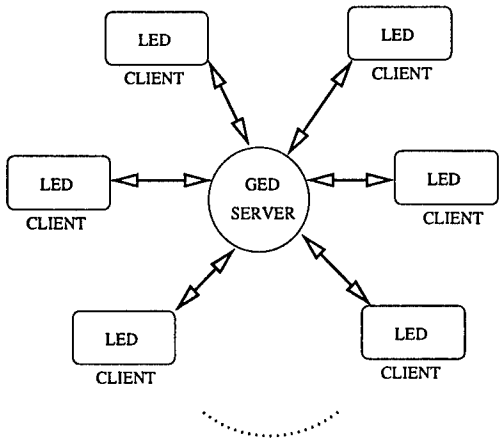


Figure 12: A Client-Server Architecture of GED

6.4.2 Architecture of Global Event Detector

Figure 13 illustrates the architecture of the Global Event Detector.

On every local site, local events are detected by the Local Event Detector, and a client application communicates with the server through a GED interface. In order to receive detections of global events, each client should register and send the necessary information (global event specification) to the server at the beginning of the process. In addition to sending an event occurrences to a local event manager (LED) for composite event detection and rule execution of ECA rules, local events need to be sent to the GED for global event detection according to the global event specification. The registration and local event notification to the server are managed by a GED interface which is an extension to LED.

On the server site, a socket interface is implemented to receive the request and send the reply message to clients. When the server receives the request from clients, it invokes corresponding services, and begins global event detection. Whenever a global event is detected by the server using the event's subscribers, the server will notify the corresponding clients along with event names and parameter lists through the socket interface.

6.4.3 Data Structures Of Global Event Detector

The data structure of GED is illustrated in Figure 14

At each client site, an Extended Local Event Detector (ELED) combined with LED are used to detect local and global events. A GED Interface that is implemented by socket mechanism is established to communicate with the GED server which includes sending event detection requests and receiving global event notifications. On the server site, Global Event Detector receives requests from client applications and records the client socket ID and application ID. It also receives event trees from clients and builds a global event graph. Whenever a global event is detected, it will propagate event notification to its parent nodes according to its subscribers, compute its param-

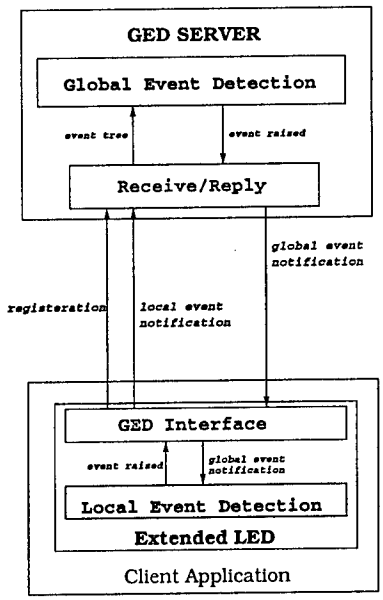


Figure 13: Global Event Detector Model

eter linked list, and send this notification along with its parameter list back to specific clients as appropriate. When a parent receives an event notification from its child (constituent) event, it will record this event instance along with its parameter list for further event detection.

6.4.4 Class Hierarchy In The Global Event Detector (GED)

As shown in Figure 15, the class hierarchy of GED is similar to that of LED.

A PRIMITIVE class in LED specifies primitive event objects that is defined by method signatures inside this application. Since global primitive events denote external events that are detected outside of the local application, the PRIMITIVE class is not useful anymore. Instead, a GLOBAL class is defined to represent the global primitive event objects. Three attributes are defined inside the GLOBAL class: *send_sname, send_ename, event_no. send_sname* denotes the application ID that this event instance needs to be notified by the server after it is raised. *send_ename* is the name of this event that application *send_sname* uses. It has the same value of *ename* attribute of a REMOTE class instance which is related to this global primitive event in application *send_sname. event_no* denotes the instance number of the occurrence of this event. To capture the global event features, two attributes are added to the NOTIFIABLE class: *site* and *send_back. site* attribute specifies the application ID where this event is defined and detected. *send_back* is a flag to indicate whether this event notification needs to be sent to any applications by the server after it is detected. Because of the time delays associate with communication and network failures, "P" and "P*" operators are not supported by GED.

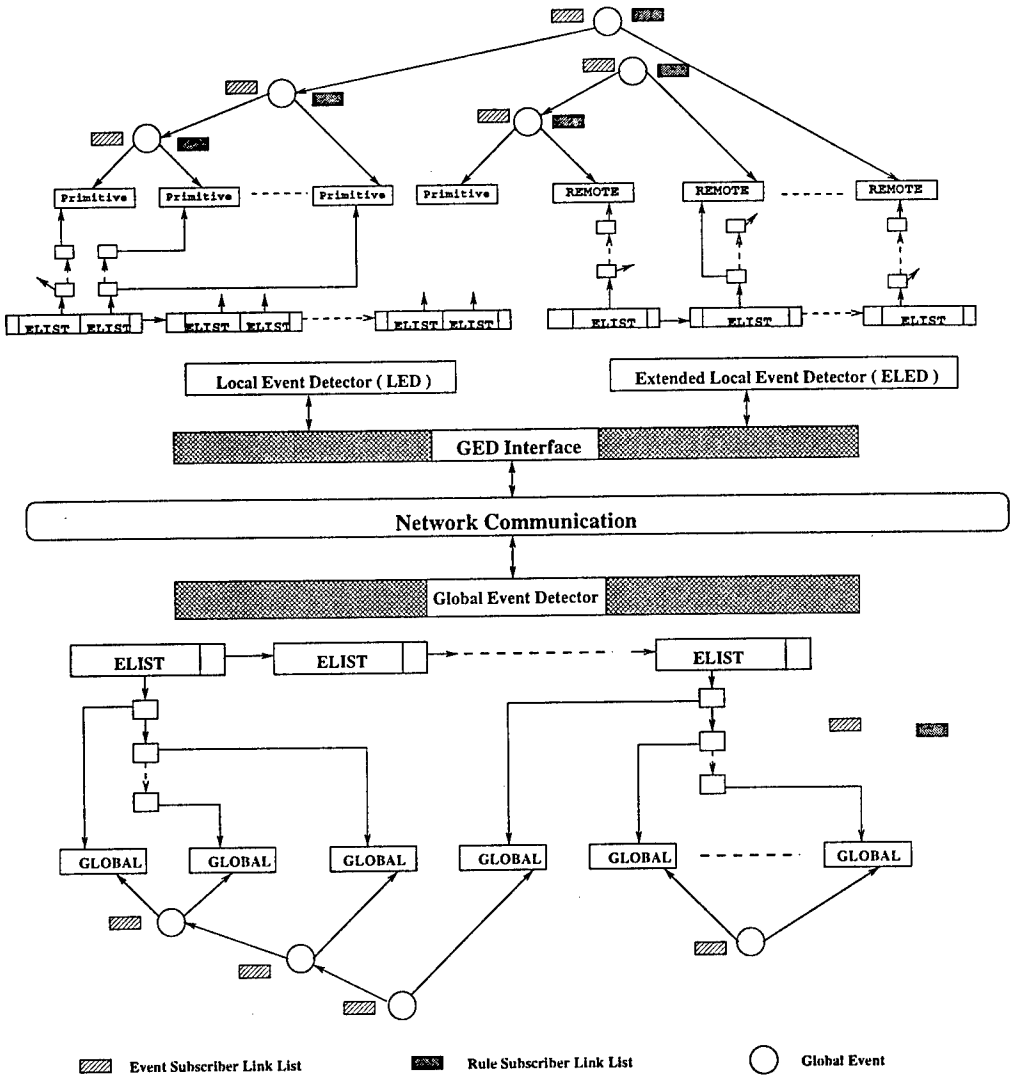


Figure 14: Data Structure of Global Event Detector

6.4.5 Event Graph

Global events are detected on the server using a event graph. An event tree is created for each composite event and these trees are merged to form an event graph for detecting a set of composite events. This will avoid the detection of common sub-events multiple times thereby reducing storage requirements. The leaf nodes are made of global primitive events, whereas the non-leaf nodes represent global composite events. For each node in the event graph, there is an event subscriber linked list containing all the composite events that use this event as its constituent one. An event node has a pointer to its subscriber which becomes its parent node. Whenever a global primitive event is detected, it will propagate the event notification to its subscribers, that is, its parent nodes. Event occurrences flow upwards as in a data-flow computation. The parent nodes maintain the occurrence of its constituent events along with their parameter lists which are stored separately for

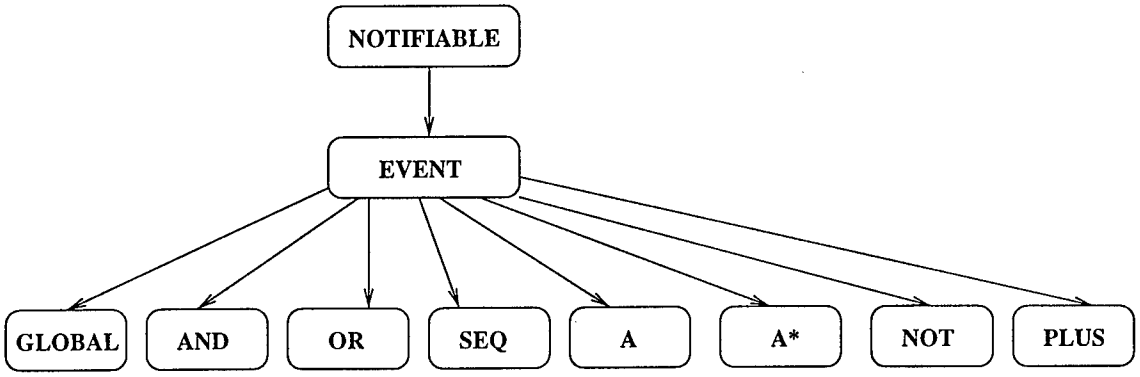


Figure 15: GED Class Hierarchy

each context set to the node. If the composite event occurs by the last notification, it is detected and further propogates to its subscribers. Each time an event is raised, it will check its "send_back" flag. If the "send_back" flag is true, the server will send this event notification to a specific application according to this event "site" attribute.

An event graph example is illustrated in Figure 16. Global primitive events G1, G2, G3,

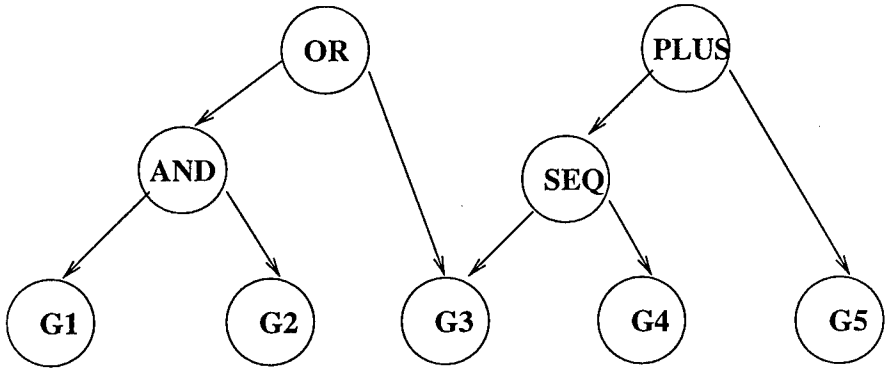


Figure 16: An Event Graph Example

G4, G5 are represented to be leaf nodes in the event graph. Four global composite events are constructed using these leaf nodes with event operators (AND, OR, SEQ, PLUS). Whenever each global primitive event is signaled, it will propagate event notification to its parent nodes (subscribers) immediately, and continue this notification upwards as appropriate. computation.

6.4.6 Global Event Detector

GED is composed by a EVNT_LIST class instance. Each remote site that is to participate in global event detection needs to register to the GED on the server, and GED creates a EVNT_NODE object for each such site. For each EVNT_NODE class object, there is a event list that records global events being detected and sent from this site. Each node in this event list is related to a leaf node of the global event graph. A composite event is constructed from these leaf nodes, and using them,

additional composite events are constructed, thus creating an event graph.

- Event nodes in event graph

  Each global event sent from client corresponds to a node in the event graph. A Global primitive event corresponds to a leaf node whereas a global composite event corresponds to an internal node. Each node has an event subscriber list and a site subscriber list.

  The event subscriber list records composite events that are related to this global event. Each node has a pointer to each of its subscribers . Thus each subscriber of a global event becomes one of its parent node that the event tree is built from.

  The site subscriber list of a global event node is a list of sites that are interested in this event. It records the site name and address. When a global event is fired, it checks its site subscriber list, and sends notification to each site on their list.

- Global event detection

  When a global event is signaled from a remote site and sent to the server, it is related to the corresponding SITE class link node, traverses this node's event list, identifies the corresponding leaf node of the event detection graph, and broadcasts the notification to its event subscribers immediately. It also goes through its site subscribers, sends the notification and parameter lists back to local site. When the event subscribers of this global event receive the notification from their child nodes, they either record this event and wait for further notification, or broadcast the notification to their event subscribers and site subscribers if the firing condition is met.

6.4.7 Communication Between A Client And The Server

Since global event detection is distributed between clients and their server, the communication between client and server is important. There are four types of messages passed between client and server.

- Global event detection request from a client to the server.

  This is done during the handshake between a client and the server. The request includes global events that need to be detected by the server. The global event graph in the server is constructed based upon these event trees sent by a client.

- A event name list from server to client.

  This event name list contains all the events that need to be detected in this site and sent to the server after this event has occurred.

  As mentioned earlier, instead of propagating every event from a local site, only those events that are used by other sites need to be notified to the server. A global event name table in each

local site is used to record such event names. Whenever a new client makes a connection to the server, the global event name table on related local sites is updated dynamically according to the name list sent from the server.

- Event notification from client to server.

  When a local event is signaled, in addition to notifying the LED, it will check the global event name table and sends the notification message to the server if it needs to.

- Event notification from server to client.

  Whenever a global event is detected in the server, it will check its site subscribers, and sends the notification message back to these sites.

6.4.8 Implementation

To satisfy the client/server model of Global Event Detector, RPC is used as the programming tool in SENTINEL. GED provides services to client applications for global event detection.

Four services are implemented on the server side to meet this requirement.

- Global_event_registration

  During the handshake with the server, a client calls this global_event_ registration service to register all the global events that need to be detected by and notified from the server. Event trees are sent to the server using which the global event graph on the server site is constructed and updated.

- Local_event_notification

  Whenever a local event is detected at a client site, it will check the global event name table, and calls the local_event_notification service if this event need to report its notification to the server according to this event name table. When the server receives the notification, it will send it to the GED immediately for further global event detection.

- Global_event_name_list_update

  Whenever a client is up and connect to the server, server will create and update a global event name list for each local site. Then it calls this
  global_event_name_list_updating service to update the global event name table in related clients.

- Global_event_notification

  Whenever a global event is detected on the server, it will check its site subscribers, and call this global_event_notification service for each subscriber. After a related client receives this global event notification, it will send the notification to LED for further event detection.

Since a client application may come up anytime, the server should send the client appropriate messages no matter when the client communicate with it. The *event name list* for this client should be stored by the server until the client gets it. So does event notifications. Global events required by this client may occur when the client is not running (the client either exits normally or is terminated by the interrupt). The server keeps all event notifications for this client application until the client restores its execution again. Since the client socket_ID recorded by the server changes when a client is terminated abnormally and comes up later, the server should keep the updated socket address for this client.

In GED, When a client connects to the server, the server calls following procedures:

- Check the client socket address table to update the socket address for this client.
- Check the *event name linked list* to see if there are events that defined by the client are required the other applications. If it does, the server will send this *event name linked list* to this client and delete this linked list. The client process then updates its local *event name linked list* according to the messages it received.
- Check the event notification linked list. If there are global events that occurred when the client was not running, the server will send these notifications (as a linked list) to the client process and then delete them from the linked list. This avoids loss of events provided that the server is always alive.

7 Conclusions and Future Work

7.1 Conclusions

This report presents an approach to monitor events in a distributed database environment. In earlier work, an event specification language Snoop, a Snoop preprocessor spp, and a local event detector were developed as part of Sentinel to define and detect events in a single address space. To monitor the event behavior in a distributed database system, a global event detector mechanism for global event detection, and an extension to Snoop language for global event definition are needed to meet this requirement.

This report proposes an approach to event-driven monitoring of distributed systems which includes the extension of Snoop and spp to support global event definition and implementation of a global event detector to detect events spanning several applications. Section 1 and section 2 describe recent work on active database management systems . Most of the earlier work do not address event specification outside of their address space.Rule cannot be specified on events that involve multiple applications. That is, one of them support processing ECA rules in a distributed environment.

Section 3 provides an overview of Snoop language and its preprocessor spp.

Section 5 presents extensions to Snoop and spp to support global event definition. A stand-alone global event definition is added to Snoop without modifying the original BNF. Snoop preprocessor *spp* is modified to support processing global event specifications and create a global event specification file for communication purposes between a client and the server.

In section 4, several alternative architectures of Global Event Detector are presented and compared, and a client/server model is introduced. In this client/server architecture, a GED server receives requests from client applications and provides services (RPC calls) to detect global event and sends event notification back to the client whenever an global event is detected. This model supports several applications running on the same machine as well as those running on the different machine.

The implementation details of GED is introduced in section 6. In client application, an Extended Local Event Detector (ELED) combined with LED and a GED Interface is used to detect events and communicate with the GED. A Global Event Detector is implemented in the server to receive event detection requests from client applications, detect global events using an event detection graph, and to send the event notification back to clients.

7.2 Future Work

- Persist events and retrieve them as needed to support client failure. This approach avoids lose of events and uses small memory spaces for event detection. Client failures can be better tolerated.

- Define rules at the GED through an interface. This can be used for propagating event notifications from one application to another. Updating data across databases can be realized in this approach.
- Use of a distributed transparent mechanism such as CORBA for generalizing the concept proposed in this report .
- Use of operator P* (as well as A and A*) at the server to propagate information from one client to the other. This can be used for asynchronous transfer of data, update propagation etc.

References

[1] U. Dayal, B. Blaustein, and A.P. Buchmann. The HiPAC project: Combining active databases and timing constraints. SIGMOD RECORD, 17(1), March 1988.

[2] N. H. Gehani, H. V. Jagadish, and O. Shmueli. COMPOSE: A System For Composite Event Specification and Detection. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, December 1992.

[3] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Unified Approach. In *Proceedings 17th International Conference on Very Large Data Bases*, Barcelona (Catalonia), Spain, Sept. 1991.

[4] S. Gatziu, K.R. Dittrich. Samos: An active object-oriented database system.. *IEEE Quarterly Bulletin on Data Engineering*, March 1993.

[5] R. Orfali, D. Harkey, and J. Edward. The Essential Distributed Objects Survival Guide. John Wiley & Sons, Inc., NJ, 1996.

[6] Object Management Group, Framingham, MA. CORBAServices: Common Object Services Specification v1.0. March 1995.

[7] S. Schwiderski. Monitoring the Behaviour of Distributed Systems. Ph.D thesis, University of Cambridge, London, 1996.

[8] D. Mishra. SNOOP: An Event Specification Language for Active Databases. Master's thesis, University of Florida, Gainesville, 1991.

[9] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1–26, October 1994.

[10] S. Chakravarthy and D. Mishra. Towards an Expressive Event Specification Language for Active Databases. In *Proceedings of the 5th International Hong Kong Computer Society Database Workshop on Next Generation Database Systems*, Kowloon Shangri-La, Hong Kong, February 1994. (Invited Paper).

[11] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *Proceedings, International Conference on Management of Data*, Washington, D.C., pages 99–108, May 1993.

[12] V. Krishnaprasad. Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation. Master's thesis, University of Florida, Gainesville, 1994.

[13] L. Hyesun. Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing. Master's thesis, University of Florida, Gainesville, 1996.

*U.S. GOVERNMENT PRINTING OFFICE: 1999-610-130-81087

MISSION OF AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.